

Les Threads en .NET

Par gretro



www.openclassrooms.com

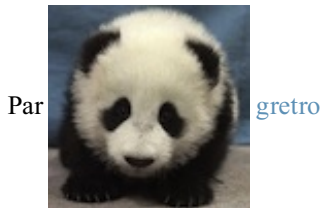
*Licence Creative Commons 4 2.0
Dernière mise à jour le 24/08/2011*

Sommaire

Sommaire	2
Lire aussi	1
Les Threads en .NET	3
Les bases des delegates	3
Les delegates, une référence en la matière !	3
Partis pour la gloire !	4
Les threads	7
Un peu d'histoire	7
Le multi-task en Windows	7
Les threads, enfin !	8
Les mécanismes de synchronisation	12
Les threads avec les Windows Forms	24
BackgroundWorker	28
Partager	30



Les Threads en .NET



Par

gretro

Mise à jour : 24/08/2011

Difficulté : Difficile



Il existe en .NET quelques créatures qui se cachent dans le noir. Parmi celles-ci sont les threads, les fonctions lambda et les delegates. N'ayez craintes chers Zéros, voici un tutoriel qui répondra à vos attentes en ce qui concerne les threads.

Par contre, l'étude de cette discipline requiert également une bonne connaissance du C#. Nous verrons également au passage les delegates, références sur des méthodes.

Je vous le dis maintenant, ce n'est pas un sujet des plus faciles, mais cela permet de nombreuses améliorations de vos applications, notamment pour leur permettre de communiquer en réseau, mais aussi de rendre fluide certaines opérations lourdes en traitement sans bloquer votre application !



Cours pour zéros avertis seulement ! Si vous ne connaissez pas le C#, ce tutoriel n'est pas pour vous.



Si vous désirez suivre le cours en VB.NET, je vous conseille d'utiliser un traducteur C# -> VB.NET afin de traduire les exemples.

Sommaire du tutoriel :



- Les bases des delegates
- Les threads
- Les threads avec les Windows Forms

Les bases des delegates

Les delegates, une référence en la matière !

Qu'est-ce qu'un delegate ?

Un delegate est un concept abstrait du C#. Jusqu'à maintenant, une variable pouvait contenir de nombreuses choses. On traite, par exemple, les objets comme des variables. Elles permettent aussi de mettre en mémoire des données, comme du texte, des nombres entiers ou flottants, des booléens. Ces cas ne sont que des exemples.

Un delegate est en fait une variable un peu spéciale... Elle ne sert qu'à donner une **référence vers une méthode ou fonction**. Elle est donc de type référence, comme un objet !

L'utilité sera bien évidemment d'envoyer ces delegates en paramètres ! Pensez-y, une méthode générique unique pourrait s'occuper de lancer un nombre infini d'opérations déterminées par vos bons soins. Ne vous en faites pas si ce concept est abstrait pour le moment. Comprenez seulement qu'on peut drastiquement augmenter la réutilisation du code avec cet outil. N'est-ce pas le grand but de la POO que de réutiliser le code ?

Comment on fait un delegate ? Ça paraît compliqué...

Oui, ça paraît compliqué aux premiers abords, mais ça devient vite facile. Allez, on s'y lance !

Partis pour la gloire !

Je vais commencer par vous donner un exemple bien simple, un cas que vous utilisez toujours lorsque vous programmez en C# !

Code : C#

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace TestThread
{
    class Program
    {
        static void Main(string[] args)
        {
            bool resultat = Test("Ceci est un test qui est négatif
!");
            bool res2 = Test("Positif");

            static public bool Test(string test)
            {
                return test.Length < 15;
            }
        }
    }
}
```



Une fonction est un bloc de traitements qui retourne un résultat, et la méthode ne fait que des traitements sans rien retourner. Pour alléger le texte, d'ici la fin de ce tutoriel, j'appellerai toute fonction ou méthode des méthodes, indifféremment de leur traitement.

On voit clairement une situation très usuelle ici. Vous appelez **Test** deux fois à partir du **Main**. Ce qui se passera, c'est que lorsque viendra le temps d'exécuter ce code, .NET lancera la méthode **Test** afin de donner un résultat à la variable *resultat*. On fait alors un appel de la méthode.

Je vais immédiatement déclarer un delegate pour la même situation, comme ça, vous verrez de quoi il en retourne.

Code : C#

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace TestThread
{
    class Program
    {
        //Mon delegate aura exactement la même signature que ma
        méthode !
        delegate bool PremierDelegate(string i);

        static void Main(string[] args)
        {
            //Je crée une variable a qui contiendra la méthode
            Test.
            PremierDelegate a = new PremierDelegate(Test);

            //Au lieu d'appeler Test, je vais appeler a, ce qui me
            donnera le
            //même résultat !
        }
    }
}
```

```

        bool resultat = a("Ceci est un test qui est négatif !");
        bool res2 = a("Positif");
    }

    static public bool Test(string test)
    {
        return test.Length < 15;
    }
}

```

Bon, dans cet exemple, l'utilisation d'un delegate est carrément inutile, mais ils deviendront rapidement indispensables, surtout lors de la programmation réseau.

Une autre utilisation très fréquente des *delegates* est la **gestion des événements** ! Quand vous vous abonnez à un événement, vous refilez tout simplement une méthode à une liste de delegate. Quand on **invoque** un événement, on appelle toutes les méthodes abonnées. Les événements peuvent faire l'objet d'un autre tutoriel et dépassent les objectifs de ce tutoriel. Si vous voulez plus d'informations, veuillez visiter le site de [MSDN](#) en attendant.



Dans cet exemple, je me réfère au concept de signature d'une méthode. Je vais prendre quelques minutes pour vous expliquer.

Les signatures de méthodes

Toute méthode possède une signature. Une signature de méthode, comme dans le monde réel, sert à identifier une méthode de façon unique. Si vous avez déjà fait de la programmation, peut-être avez-vous déjà vu ce concept.

La signature de méthode résout le problème des noms uniques dans le cas de surcharge d'une méthode. Dans les langages autorisant la surcharge de méthode, on se réfère à la signature plutôt qu'au nom de la méthode pour l'appeler. Pour chaque méthode doit correspondre une signature différente des autres. Je vais vous faire un cours en accéléré, donc je vous invite à vous référer à la documentation de MSDN ou alors au Forum du Site du Zéro si vous avez de plus amples questions.

Donc, la signature d'une méthode contient les informations suivantes :

- L'identificateur de la méthode
- La séquence des types de la liste des paramètres



Le type de retour ne fait pas partie de la signature ! En effet, ce qui est important de retenir est le Nom de la méthode et les paramètres dans l'ordre.

Ce qui fait que notre méthode `static public int Test(string test)` a la signature suivante : `Test(string)` ;

La **définition** d'un delegate est un peu différente. Le nom de la méthode et des paramètres sont inutiles. Ce qui compte c'est le type de retour et l'ordre des paramètres selon leur type (pas leur nom). Ainsi, un delegate ne pourra référer qu'une méthode possédant la même **définition**.

L'autopsie d'un delegate !

Analysons maintenant comment créer ce fameux *delegate*. Tout d'abord, sachez que le rôle principal d'un *delegate* est de passer, croyez-le ou non, une méthode en paramètre à une autre méthode. On peut alors aisément imaginer la flexibilité d'un code en permettant à une méthode générique d'appeler une méthode passée en paramètre et d'en afficher le résultat, peu importe les opérations à effectuer. Nous verrons un exemple un peu plus tard. Pour le moment, voici la définition d'un *delegate* :

[Attribut] [Modificateur d'accès] delegate typeRetour NomDuDelegate (paramètres)



Les cases entourées de crochets sont optionnels. Un **attribut**, par exemple, sert à injecter des *meta-données* récupérables à l'exécution du programme. Cela dépasse le cadre de ce cours, mais vous pouvez vous rabattre sur



MSDN pour plus d'explications. Les **modificateurs d'accès**, quant à eux, sont une notion qui devrait vous être connue. Il s'agit de modifier l'accès à l'aide des mots clés **public**, **private**, **protected**, **internal**, etc.

Voici un exemple :

```
class Program
{
    delegate int Calcul(int i1, int i2);

    static void Main(string[] args)
    {
    }
}
```

Je parlais un peu plus haut de la définition d'un **delegate**. Il faut bien la lui donner cette définition ! On fait généralement cela dans le même espace où l'on déclare les *variables globales*. Vous verrez dans l'exemple qui suit. Ensuite, on utilise le *delegate* comme un **objet**. On peut alors l'utiliser dans les **paramètres** ou ailleurs si nécessaire. Ce sera plus clair pour vous avec l'exemple qui suit :

Code : C#

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace TestThread
{
    class Program
    {
        delegate int Calcul(int i1, int i2);

        static void Main(string[] args)
        {
            //Affichage de la console.
            Console.WriteLine("Test de delegate");
            Console.WriteLine("-----");

            //On passe à la méthode Afficher la méthode à lancer et
            les arguments.
            Afficher(Add, 25, 19);
            Afficher(Sub, 52, 17);
            Afficher(Mul, 10, 12);
            Afficher(Div, 325, 5);

            //On ne ferme pas la console immédiatement.
            Console.ReadKey();
        }
    }
}
```

```
//On fait une méthode générale qui prendra le delegate
en paramètre.
static void Afficher(Calcul calcul, int i, int j)
{
    Console.WriteLine("{0} {1} {2} = {3}", i,
calcul.Method.Name,
j, calcul(i, j));
}

//Méthodes très simples qui ont toutes un type de retour et
des paramètres identiques.
static int Add(int i, int j) { return i + j; }
static int Sub(int i, int j) { return i - j; }
static int Mul(int i, int j) { return i * j; }
static int Div(int i, int j) { return i / j; }
}
```

C'est pas du chinois, mais c'est pas simple, hein ? En effet, ça arrache une grimace la première fois qu'on voit ça, mais en fait, c'est très simple. Comme toutes nos méthodes répondent à la même **définition** que notre *delegate*, nous sommes en mesure de toutes les utiliser à l'aide du même. C'est un peu comme de dire qu'un *int* peut égaler 0 ou bien 12154, car les deux répondent à la même définition, soit être un entier entre **int.Min** et **int.Max**.

Ce code, bien que simple est très puissant. Si je voulais ajouter l'opération modulo, il serait TRÈS TRÈS simple de le faire, vous ne trouvez pas ?

Cela conclut notre introduction aux *delegates*. Il y en a plus que ça à savoir et à comprendre, mais si vous maîtrisez cette partie, c'est excellent. Si vous ne maîtrisez pas bien, je vous recommande de la relire ! Si vous avez fait du C ou du C++, cette fonctionnalité ressemble étrangement aux pointeurs sur fonction. Lorsque l'on fera de la programmation réseau, ces *delegates* deviendront essentiels ! Courage, c'était vraiment le plus difficile ...

Les threads

Un peu d'histoire

On parle beaucoup de threads ces temps-ci. Les nouveaux processeurs sont des puces conçues afin d'optimiser le traitement de plusieurs threads simultanés. Il y a quelques années, à l'ère du Pentium 4, on ne cessait d'augmenter la fréquence d'horloge afin d'optimiser la vitesse d'un seul coeur. Chaque thread avait un numéro et attendait son tour afin d'être traité.

Ne vous méprenez pas, cela n'a pas changé, mais les processeurs ont commencé à les traiter simultanément, d'abord avec la technologie HT sur les derniers Pentium 4, puis à l'aide de multiples coeurs avec les Core 2 Duo / Quad. Maintenant, il s'agit d'un mélange des deux, soit de multiples coeurs qui appliquent chacun une technologie HT, comme dans le Core i7 d'Intel. Pardonnez-moi, mais je connais très mal les processeurs AMD, étant un utilisateur d'Intel majoritairement 😊.

Cela explique un peu l'évolution de la technique de traitement des threads, mais je ne vous ai toujours pas expliqué comment fonctionne le multi-task en Windows.

Le multi-task en Windows

Un ordinateur, ça ne sait faire qu'une seule chose à la fois !

Windows, comme tout bon SE actuel se sert d'une méthode particulière afin de simuler un multi-task. En effet, un processeur ne sait que faire une chose à la fois. La technique est bien simple, il s'agit de créer un système de jeton et de le passer à chaque processus pour un certain temps selon leur priorité.

En ce moment même, vous utilisez votre navigateur préféré pour visiter le site du Zéro, mais cela n'empêche pas votre ordinateur de vaquer à d'autres occupations. Par exemple, vous êtes peut-être en train de copier un fichier, ou même juste en train d'avoir 3 fenêtres ouvertes sur le Bureau en ce moment. Windows doit rafraîchir leur contenu à toutes les x millisecondes afin de créer un sentiment de fluidité chez l'utilisateur.

Donc, suivant cet exemple, Windows aura un jeton à accorder à votre navigateur web pour tant de temps, puis suspendra ses calculs et opérations et donnera le jeton à un autre traitement. Lorsqu'il reviendra au navigateur, celui-ci sera autorisé à continuer

ses opérations. Comme ça, tout le monde est content, mais surtout l'utilisateur qui désire ouvrir Google Chrome en même temps que MSN et Word, ainsi que Visual Studio 2010. Ne riez pas, c'est pas mal le scénario actuel de mon PC en ce moment... Tout ça pour dire qu'on commence la section sur les threads pour de vrai !

Les threads, enfin !

Les threads sont des exécutions que l'on sépare de l'exécution principale pour les raisons suivantes :

- Tâche exigeante (gros calculs, gros traitements, etc)...
- Tâche détachée (impression, recherche, etc)...
- Tâche bloquante (ça sent le réseau ici !)

Créer un thread dans ces cas est utile afin de créer un certain parallélisme dans les exécutions.



Les threads rendent un code beaucoup plus complexe, non seulement à l'écriture, mais aussi au débogage ! Oh là là, que de frustration passées à programmer avec les threads. Cela dit, ils apportent énormément de puissance à une application ! Cette complexité vient du fait que le système de jeton est imprévisible ! Par exemple, il pourrait passer 3 secondes sur un thread A, mais 2 secondes sur un thread B. Bien-sûr, on ne parle pas de secondes ici, mais bien de nano-secondes. Un autre point frustrant sera que lors du débogage, vous pourriez vous retrouver dans une méthode complètement différente de celle qui vous intéresse en un clin d'oeil, justement à cause que le système de jeton a changé de thread et vous a projeté dans la méthode qu'il exécute au moment même.

Le cas des tâches bloquantes...

Une application Windows Forms est amenée à se rafraîchir assez fréquemment. Lors de grosses opérations ou d'opérations synchrones qui bloquent, tout le temps de calcul est alloué à ces tâches et non plus au rafraîchissement. Après un certain temps, Windows déclare l'application comme "Ne répondant plus". Si vous planifiez de distribuer votre application, ce comportement est inacceptable, vous en conviendrez. C'est dans ce type de cas qu'on utilisera les threads. Imaginez afficher une belle animation sur un *Splash Screen* alors que les ressources sont en chargement en arrière plan.

Les différents types de thread

Il est possible de créer deux types de threads, bien que cela revienne au même. Lors de l'instanciation de la classe `Thread`, il est possible de garder la référence de l'objet, ou de la laisse flotter. Si on ne la récupère pas, on appellera ce thread "indépendant". Il sera créé, puis lancé immédiatement. Comme on ne gardera pas la référence, on ne pourra pas contrôler ce thread du tout. Il fera ce qu'il a à faire, sans que l'on puisse intervenir (sauf en utilisant quelques primitives de synchronisation que nous verrons plus tard).

Voici comment déclarer un thread indépendant :

Code : C#

```
new Thread(fonction).Start();
```



Puisqu'il n'y a pas d'opérateur d'affectation (=), on laisse partir la référence. Lorsque le thread se terminera, le Garbage Collector passera en arrière et se débarrassera de l'objet pour nous.

Le type dépendant est beaucoup plus fréquent. Il s'agit de garder la référence sur l'objet afin de pouvoir l'analyser, le tester, l'influencer. Vous connaissez déjà tous comment le créer, mais pour la forme, voici un exemple :

Code : C#

```
Thread nomThread = new Thread(fonction);
```



Cela peut porter à confusion, mais tous les threads d'une même application sont appelés *Processus* dans l'environnement Windows. Ceci provient du fait que Windows désire grouper tous les threads appartenant à une



application au cas où ça tournerait mal. Cependant, le système de jeton mentionné ci-haut fonctionne au niveau des threads lui-même, et pas des processus. Cela signifie également que de mettre fin au thread principal met en effet fin aux threads enfants, indépendants ou dépendants, car on détruit carrément le processus et les threads qui le composent.

Comment lancer le thread ?

Tout d'abord, assurez-vous d'utiliser l'espace de nom (vous savez, les *using* tout en haut de votre fichier .cs) **using System.Threading;**

Déclarer un nouveau thread va comme suit :

Code : C#

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading;

namespace TestThread
{
    class Program
    {
        static void Main(string[] args)
        {
            //On initialise l'object Thread en lui passant la
            //à exécuter dans le nouveau thread. Ça vous rappelle
            //certains delegates ça ?
            Thread th = new Thread(Afficher);

            //Un thread, ça ne part pas tout seul. Il faut lui
            //commencer l'exécution.
            th.Start();

            Console.ReadKey();
        }

        static void Afficher()
        {
            //Code tout bête qui affiche la lettre A 1000 fois.
            for (int i = 0; i < 1000; i++)
            {
                Console.Write("A");
            }
        }
    }
}
```

Ce code fonctionne bien dans le cas où on n'a aucun paramètre à passer. Il est un peu plus compliqué d'en passer, mais on s'en sort, vous verrez.

Code : C#

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading;

namespace TestThread
{
    class Program
    {
```

```

1
static void Main(string[] args)
{
    //Il faut créer un objet ParameterizedThreadStart dans
    le constructeur
    //du thread afin de passer un paramètre.
    Thread th = new Thread(new
    ParameterizedThreadStart(Afficher));

    Thread th2 = new Thread(new
    ParameterizedThreadStart(Afficher));

    //Lorsqu'on exécute le thread, on lui donne son
    paramètre de type Object.
    th.Start("A");

    th2.Start("B");

    Console.ReadKey();
}

//La méthode prend en paramètre un et un seul paramètre de
type Object.
static void Afficher(object texte)
{
    for (int i = 0; i < 10000; i++)
    {
        //On écrit le texte passer en paramètre. N'oubliez
        pas de le caster
        //car il s'agit d'un type Object, pas String.
        Console.Write((string)texte);

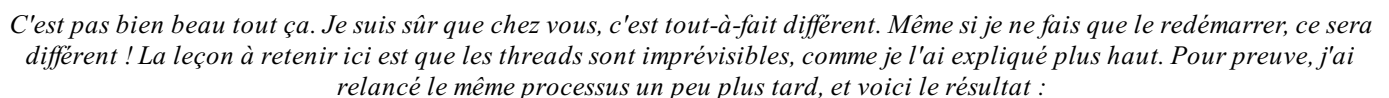
        Console.WriteLine("<-----Thread {0} terminé-----
        ----->", (string)texte);
    }
}

```



Attention, la plus fréquente source d'erreur lors de l'utilisation de ce genre de thread est le paramètre. En effet, ce type de **delegate** a sa propre définition et il requiert un seul et unique paramètre qui sera de type *object*. Faites bien attention à **trans typer (cast)** vos variables correctement !

Cet exemple est parfait pour vous montrer comment les threads sont imprévisibles, et c'est ce qui les rend compliqués ! Je vous montre le résultat chez moi.





On voit très bien que dans ce cas-ci, le Thread B a terminé en premier, ce qui prouve que le même code peut générer des résultats différents d'une exécution à l'autre, s'il est codé avec des threads !

Cas particuliers

Même si un thread s'exécute en deçà de votre programme principal, il reste que la méthode qu'il exécute fait partie de la classe à laquelle la méthode appartient. Cela signifie que l'accès aux variables globales et membres de votre classe lui seront accessibles sans problème.

Là où le problème se pose, c'est lorsque plusieurs threads devront accéder à la même variable, y faire des changements et des tests. Imaginez que votre thread A accède aux variables **nominateur** et **dénominateur** qui sont globales (à proscrire, mais bon). Le thread A a le temps de faire quelques tests, à savoir vérifier si le dénominateur n'est pas égal à zéro avant de procéder à une division. Tous les tests passent, mais juste au moment où le thread arrive pour effectuer l'opération, le thread B s'empare du jeton. Le thread B est chargé de réinitialiser le dénominateur à 0, et c'est ce qu'il fait. À ce moment là, le jeton revient au thread A qui tente d'effectuer la division. Oops, ça plante... C'est ce qu'on appelle un problème de synchronisation. Je ne vais pas vous mentir, ces problèmes sont rares. Il faut vraiment que vous soyez malchanceux. Il reste cependant important de bien synchroniser ses threads, surtout si l'on aspire à commercialiser le produit. Ainsi, plusieurs structures de synchronisation existent, et nous allons en survoler quelques unes.

Les mécanismes de synchronisation

Les variables de contrôle

Il peut sembler que les variables de contrôle soient un concept très poussé, mais pas du tout ! Il s'agit bêtement d'une variable globale que seul le thread principal modifiera et que les threads enfants contrôleront. Ce concept est particulièrement efficace dans le cas où le thread effectue une boucle infinie. Encore un fois, ça sent la programmation réseau ici. Je vous illustre le concept à l'aide d'un bête exemple.

Code : C#

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading;

namespace VarControle
{
    class Program
```



```

{
    //Quelques variables à portée globale.
    private static bool _quitter = false;
    private static int _identificateur = 0;

    static void Main(string[] args)
    {
        Console.Title = "Variables de contrôle";

        //On crée un tableau de threads.
        Thread[] threads = new Thread[5];

        //On itère à travers le tableau afin de créer et lancer
les threads.
        for(int i = 0; i < threads.Length; i++)
        {
            //Création et lancement des threads.
            threads[i] = new Thread(OperThread);
            threads[i].Start();

            //On laisse passer 500ms entre les création de
thread.
            Thread.Sleep(500);
        }

        //On demande à ce que tous les threads quittent.
        _quitter = true;

        Console.ReadKey();
    }

    static void OperThread()
    {
        //On donne au thread un identificateur unique.
        int id = ++_identificateur;

        Console.WriteLine("Début du thread {0}", id);

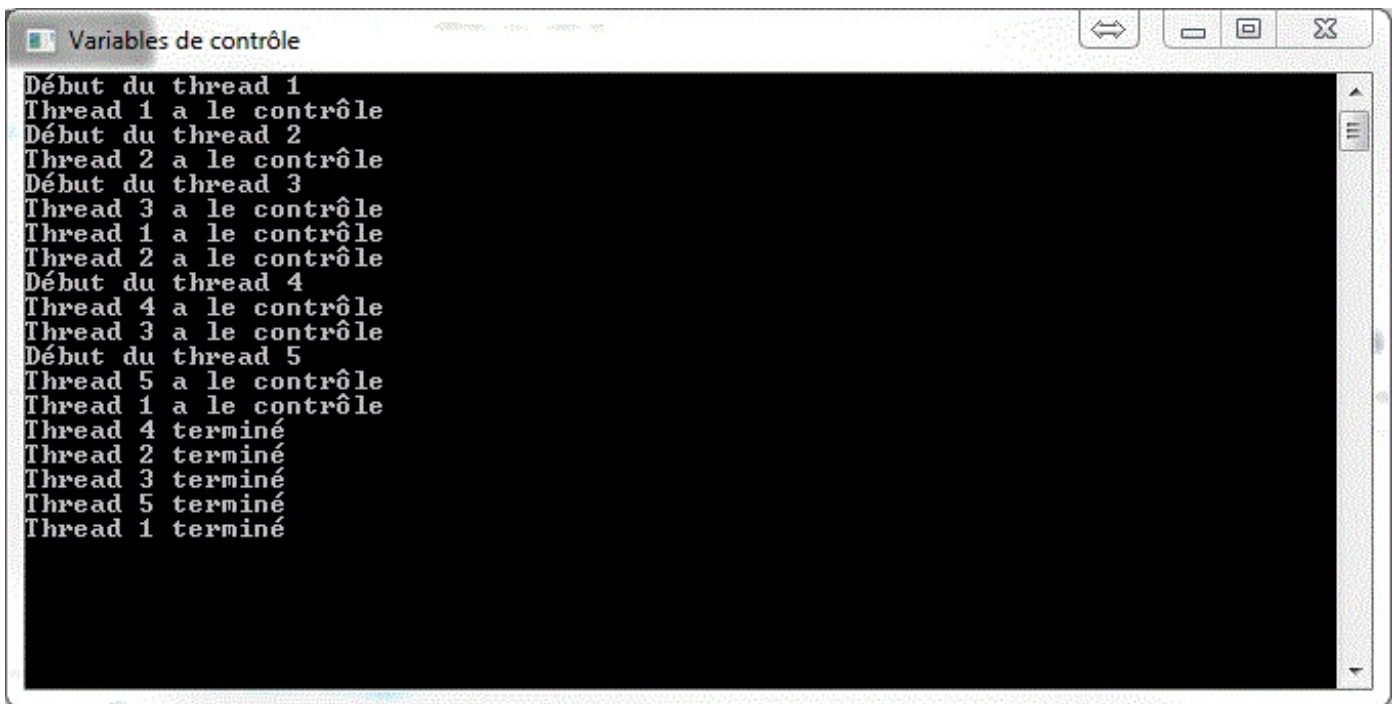
        while (!_quitter)
        {
            //On fait des choses ici tant qu'on ne désire pas
quitter...
            Console.WriteLine("Thread {0} a le contrôle", id);

            //On met le thread en état de sommeil pour 1000ms /
1s.
            Thread.Sleep(1000);
        }

        Console.WriteLine("Thread {0} terminé", id);
    }
}

```

Voici le résultat :



Ce qu'il faut comprendre de cet exemple, c'est que les variables de contrôle sont une bonne méthode afin d'influencer le comportement d'un thread, mais généralement seulement lorsque celui-ci est en boucle. Aussi, il est TRÈS important de retenir que seul le thread principal doit modifier la valeur de cette variable. Sinon, on pourrait retrouver des threads à toutes les sauces. Imaginez qu'à chaque itération, la boucle change la valeur de la variable de contrôle. On ne sait pas quand ou comment cela se produira et les problèmes feraient probablement vite leur apparition. Somme toute, il s'agit d'une bonne méthode à ne pas utiliser à outrance.



Avez-vous remarqué un bout de code qui ne vous semblait pas thread-safe ? Si oui, vous comprendrez certainement l'utilité du prochain mécanisme de synchronisation.

Secret (cliquez pour afficher)

Code : C#

```
//On donne au thread un identificateur unique.  
int id = ++_identificateur;
```

Ce bout de code n'est pas thread-safe, car on ne sait pas si un autre processus pourrait prendre le contrôle au mauvais moment. Si l'ordre de lancement est très important, cette ligne pourrait ne pas s'exécuter à temps.

Le lock

L'instruction *lock* permet de verrouiller efficacement une ressource tant et aussi longtemps qu'un bloc d'instruction est en cours. Cela signifie que si d'autres threads tentent d'accéder à la même ressource en même temps, ils ne pourront pas. Cela ne signifie pas qu'ils planteront et se termineront, mais plutôt qu'ils passeront le jeton à un autre thread et attendront patiemment leur tour afin d'accéder à cette ressource.

Voici un bel exemple :

Code : C#

```
using System;  
using System.Collections.Generic;  
using System.Linq;
```

```

using System.Text;
using System.Threading;

namespace Lock
{
    class Program
    {
        //Variable témoin du lock.
        private static Object _lock = new Object();

        //Sert à initialiser des valeurs pseudos-aléatoires.
        private static Random _rand = new
Random((int)DateTime.Now.Ticks);

        //Variable de contrôle.
        private static bool _quitter = false;

        //Variables globales étant affectées par les threads.
        private static int _nominator;
        private static int _denominator;

        static void Main(string[] args)
        {
            Console.Title = "Démonstration des lock";

            //On crée les threads.
            Thread init = new Thread(Initialiser);
            init.Start();

            Thread reinit = new Thread(Reinitialiser);
            reinit.Start();

            Thread div = new Thread(Diviser);
            div.Start();

            //On les laisse travailler pendant 3 seconde.
            Thread.Sleep(3000);
            //Puis on leur demande de quitter.
            _quitter = true;

            Console.ReadKey();
        }

        private static void Initialiser()
        {
            //Boucle infinie contrôlée.
            while (!_quitter)
            {
                //On verouille l'accès aux variables tant que l'on
a pas terminé.
                lock (_lock)
                {
                    //Initialisation des valeurs.
                    _nominator = _rand.Next(20);
                    _denominator = _rand.Next(2, 30);
                }

                //On recommence dans 250ms.
                Thread.Sleep(250);
            }
        }

        private static void Reinitialiser()
        {
            //Boucle infinie contrôlée.
            while (!_quitter)
            {
                //On verouille l'accès aux variables tant que l'on
a pas terminé.
                lock (lock)

```

```

        {
            //Réinitialisation des valeurs.
            _nominator = 0;
            _denominator = 0;
        }

        //On recommence dans 300ms.
        Thread.Sleep(300);
    }

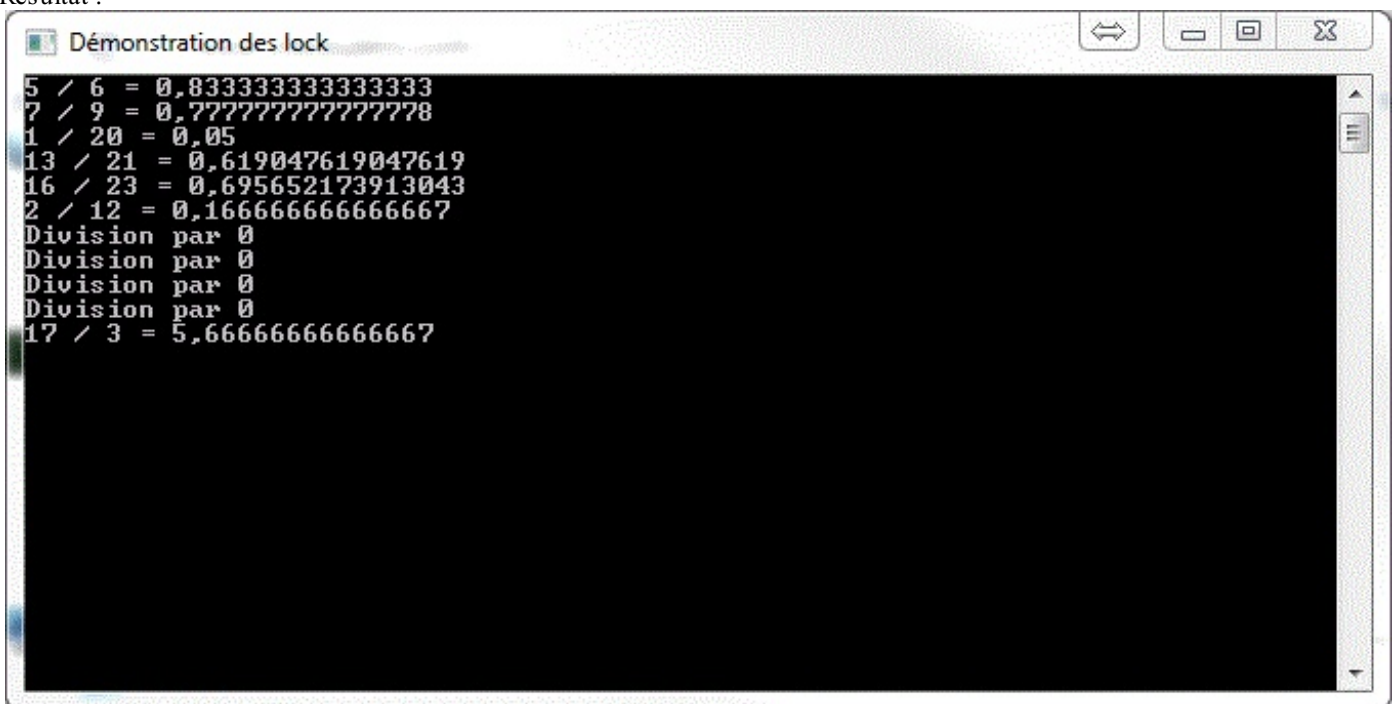
    }

    private static void Diviser()
    {
        //Boucle infinie contrôlée.
        while (!_quitter)
        {
            //On verouille pendant les opérations.
            lock (_lock)
            {
                //Erreur si le dénominateur est nul.
                if (_denominator == 0)
                    Console.WriteLine("Division par 0");
                else
                {
                    Console.WriteLine("{0} / {1} = {2}",
                        _nominator, _denominator, _nominator / (double)_denominator);
                }
            }

            //On recommence dans 275ms.
            Thread.Sleep(275);
        }
    }
}

```

Résultat :



```

Démonstration des lock
5 / 6 = 0,833333333333333
7 / 9 = 0,777777777777778
1 / 20 = 0,05
13 / 21 = 0,619047619047619
16 / 23 = 0,695652173913043
2 / 12 = 0,166666666666667
Division par 0
Division par 0
Division par 0
Division par 0
17 / 3 = 5,66666666666667

```

C'est bien comique parce que lorsque je vous ai préparé cet exemple, l'erreur dont je vous ai mentionné plus tôt s'est produite. Je n'avais alors pas englobé mon test du dénominateur dans l'instruction lock, ce qui a permis au thread en charge de réinitialiser les valeurs d'embarquer. Cela a produit une erreur de type NaN (Non-Numérique).



Malheureusement, sur le coup, je n'ai pas fait de capture d'écran, et j'ai donc essayé de reproduire l'erreur, mais sans succès. C'est donc vous dire qu'avec les threads, il faut tout prévoir dès le départ, car l'apparition d'une erreur peut être un heureux (ou malheureux) hasard !

Donc, dans cet exemple, on voit que tout est bien protégé. Aucun thread ne peut venir interférer avec les autres. Remarquez la création d'une instance d'un objet de type `Object` à la ligne 12. Cela est notre témoin de verrouillage. En réalité, n'importe quel objet qui se passe en référence peut servir de témoin de verrouillage. Comme nous avons travaillé avec des `int` dans cet exemple et que ce type est passé par valeur, nous avons eu à créer cette variable.

Voici un exemple où une variable témoin est inutile :

Code : C#

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading;

namespace lockEx
{
    class Program
    {
        static List<string> liste = new List<string>();

        static void Main(string[] args)
        {
            for (int i = 0; i < 6; i++)
                new Thread(Ajouter).Start();
        }

        static void Ajouter()
        {
            lock(liste)
                liste.Add("abc");
        }
    }
}
```

Ici, on utilisera donc l'objet `liste` qui se passe par référence, et qui est donc acceptable.



Il est possible de créer un `lock` en lui spécifiant un nom de type `string`. Cependant, Microsoft ne le recommande pas, car cette notation pourrait amener de la confusion. N'oubliez pas non plus qu'il est possible de faire de multiples `lock` qui ne protègent pas les mêmes ressources (indépendants les uns des autres). Il suffit de changer la variable témoin pour accommoder la situation.

Les Mutex

Les Mutex sont excessivement similaires aux `lock`. Cependant, si vous désirez créer de nombreuses sections critiques indépendantes, les Mutex ont l'avantage d'être sous forme d'objets plutôt que d'instructions. Un petit exemple vous éclaira sur l'utilisation des Mutex.

Code : C#

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading;

namespace MutexEx
{

```

```

class Program
{
    private const int TAILLE_TABLEAU = 2;

    //On crée les Mutex.
    private static Mutex _muxMultiplier = new Mutex();
    private static Mutex _muxDiviser = new Mutex();

    //On crée les tableaux de valeurs.
    private static int[] _valDiv = new int[TAILLE_TABLEAU];
    private static int[] _valMul = new int[TAILLE_TABLEAU];

    //Objet Random et variable de contrôle.
    private static Random _rand = new
Random((int)DateTime.Now.Ticks);
    private static bool _quitter = false;

    static void Main(string[] args)
    {
        Console.Title = "Exemple de Mutex";

        //On crée et on démarre les threads.
        Thread init = new Thread(Initialiser);
        init.Start();

        Thread mul = new Thread(Multiplier);
        mul.Start();

        Thread div = new Thread(Diviser);
        div.Start();

        //On laisse les threads fonctionner un peu...
        Thread.Sleep(3000);
        //On demande à ce que les opérations se terminent.
        _quitter = true;

        Console.ReadKey();
    }

    private static void Initialiser()
    {
        while (!_quitter)
        {
            //On demande au thread d'attendre jusqu'à ce qu'il
ait le contrôle sur les Mutex.
            _muxMultiplier.WaitOne();
            _muxDiviser.WaitOne();

            for (int i = 0; i < TAILLE_TABLEAU; i++)
            {
                //On assigne au tableau de nouvelles valeurs.
                _valMul[i] = _rand.Next(2, 20);
                _valDiv[i] = _rand.Next(2, 20);
            }

            Console.WriteLine("Nouvelles valeurs !");

            //On relâche les Mutex
            _muxDiviser.ReleaseMutex();
            _muxMultiplier.ReleaseMutex();

            //On tombe endormi pour 100ms.
            Thread.Sleep(100);
        }
    }

    private static void Multiplier()
    {
        while (!_quitter)
        {

```

```
        //On demande le Mutex de multiplication.
        _muxMultiplier.WaitOne();

        //On multiplie.
        Console.WriteLine("{0} x {1} = {2}", _valMul[0],
        _valMul[1], _valMul[0] * _valMul[1]);

        //On relâche le Mutex.
        _muxMultiplier.ReleaseMutex();

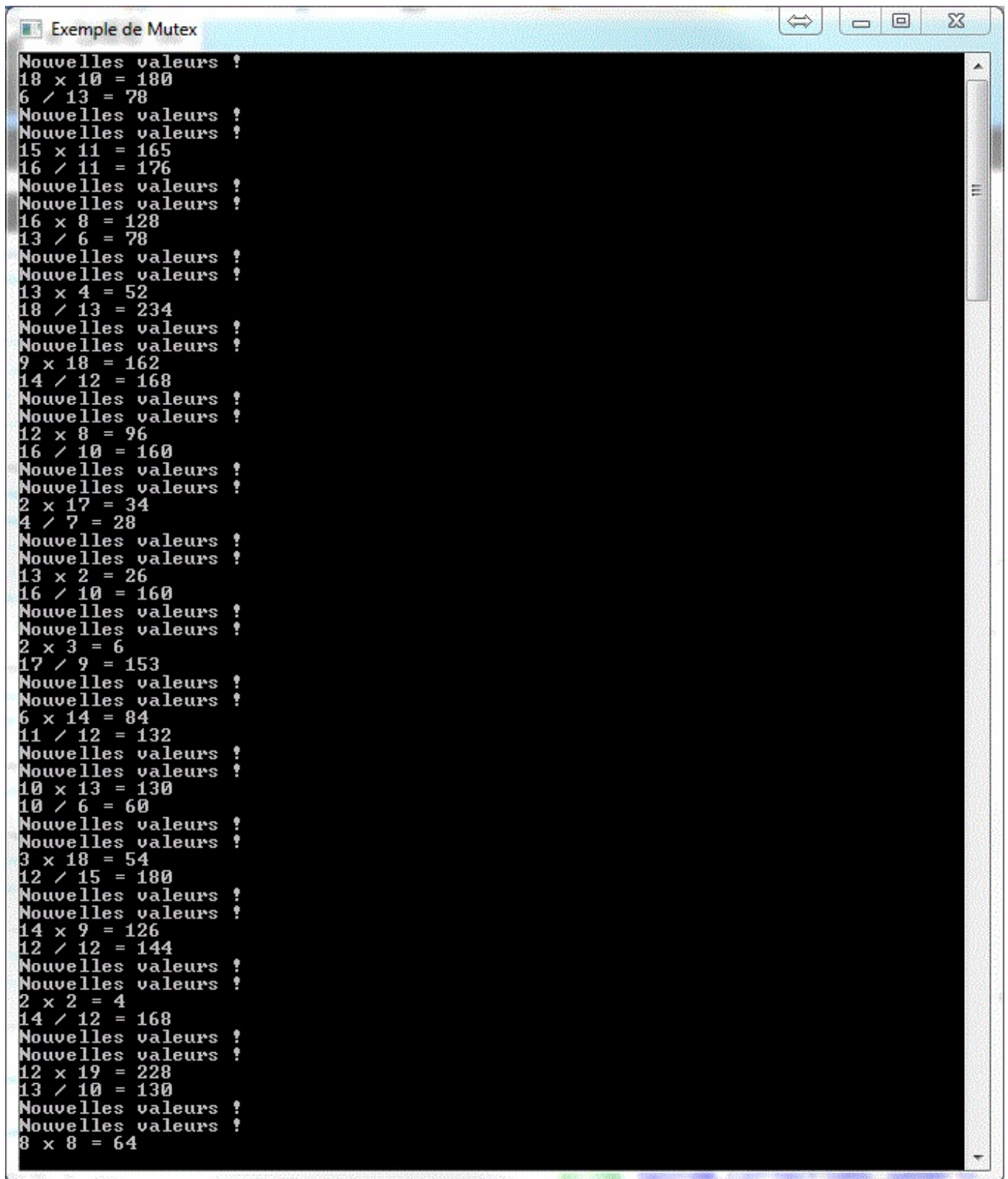
        //On tombe endormi pour 200ms.
        Thread.Sleep(200);
    }
}

private static void Diviser()
{
    while (!_quitter)
    {
        //On demande le Mutex de division.
        _muxDiviser.WaitOne();

        //On divise.
        Console.WriteLine("{0} / {1} = {2}", _valDiv[0],
        _valDiv[1], _valDiv[0] * _valDiv[1]);

        //On relâche le Mutex de Division.
        _muxDiviser.ReleaseMutex();

        //On tombe endormi pour 200ms.
        Thread.Sleep(200);
    }
}
}
```



```
Exemple de Mutex
Nouvelles valeurs ?
18 x 10 = 180
6 / 13 = 78
Nouvelles valeurs ?
Nouvelles valeurs ?
15 x 11 = 165
16 / 11 = 176
Nouvelles valeurs ?
Nouvelles valeurs ?
16 x 8 = 128
13 / 6 = 78
Nouvelles valeurs ?
Nouvelles valeurs ?
13 x 4 = 52
18 / 13 = 234
Nouvelles valeurs ?
Nouvelles valeurs ?
9 x 18 = 162
14 / 12 = 168
Nouvelles valeurs ?
Nouvelles valeurs ?
12 x 8 = 96
16 / 10 = 160
Nouvelles valeurs ?
Nouvelles valeurs ?
2 x 17 = 34
4 / 7 = 28
Nouvelles valeurs ?
Nouvelles valeurs ?
13 x 2 = 26
16 / 10 = 160
Nouvelles valeurs ?
Nouvelles valeurs ?
2 x 3 = 6
17 / 9 = 153
Nouvelles valeurs ?
Nouvelles valeurs ?
6 x 14 = 84
11 / 12 = 132
Nouvelles valeurs ?
Nouvelles valeurs ?
10 x 13 = 130
10 / 6 = 60
Nouvelles valeurs ?
Nouvelles valeurs ?
3 x 18 = 54
12 / 15 = 180
Nouvelles valeurs ?
Nouvelles valeurs ?
14 x 9 = 126
12 / 12 = 144
Nouvelles valeurs ?
Nouvelles valeurs ?
2 x 2 = 4
14 / 12 = 168
Nouvelles valeurs ?
Nouvelles valeurs ?
12 x 19 = 228
13 / 10 = 130
Nouvelles valeurs ?
Nouvelles valeurs ?
8 x 8 = 64
```

Si vous avez fait un peu de programmation en Win32 (langage C), vous pouvez voir la lignée directe des Mutex du .NET et des `CRITICAL_SECTION` du Win32. Sinon, vous voyez que les Mutex ont la même fonction que l'instruction **lock** en un peu plus verbeux. Je tiens cependant à vous avertir que de ne pas relâcher un Mutex peut faire planter votre application, donc faites attention à cela.

SemaphoreSlim

Le `SemaphoreSlim` sert à contrôler l'accès d'une ressource limitée. Jusqu'à maintenant, les mécanismes de synchronisation dont nous avons parlé ont surtout servi à limiter une ressource à un accès mutuellement exclusif entre des threads concurrents. Quant est-il si l'on veut partager une ressource, mais à travers plusieurs threads simultanément ? Cependant, on aimerait garder un

nombre maximal d'accès concurrent à la ressource. Les sémaphores existent pour cette raison. En C# .NET, il existe deux types de sémaphores. Le classique Semaphore et le SemaphoreSlim. La différence provient de la complexité de l'objet et des mécanismes internes. Le Semaphore utilise un wrapper autour de l'objet Semaphore du Win32 et rend donc disponible ses fonctionnalités en .NET. Le SemaphoreSlim, lui, est plutôt utilisé lors de courtes durées d'attente et utilise les mécanismes propres au CLR.

Je ne montrerai que le SemaphoreSlim, les deux se ressemblant beaucoup. Cependant, le SemaphoreSlim reste le plus facile et le plus léger à implémenter. Pour plus d'information sur la différence, veuillez lire cet article sur [MSDN](#). Peu importe la version qui est choisi, vous pouvez voir les Sémaphores comme un "doorman" dans une boîte de nuit. La place à l'intérieur est limitée et le doorman devra contrôler l'accès à la ressource.

Code : C#

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading;

namespace SemaphoreSlimEx
{
    class Program
    {
        //Déclaration du SemaphoreSlim qui prendra en paramètre le
        nombre de places disponibles.
        static SemaphoreSlim doorman = new SemaphoreSlim(3);

        static void Main(string[] args)
        {
            Console.Title = "Exemple de SemaphoreSlim";

            //Création des threads.
            for (int i = 0; i < 10; i++)
                new Thread(Entrer).Start(i);
            Console.ReadKey();
        }

        static void Entrer(object n)
        {
            Console.WriteLine("La personne #{0} veut entrer", n);

            //Le doorman attendra qu'il y ait de la place.
            doorman.Wait();
            Console.WriteLine("#{0} vient d'entrer dans le bar", n);
            Thread.Sleep((int)n * 1000);
            Console.WriteLine("#{0} a quitté le building !", n);

            //Le doorman peut maintenant faire entrer quelqu'un
            doorman.Release();
        }
    }
}
```

```

Exemple de SemaphoreSlim
La personne #2 veut entrer
La personne #0 veut entrer
#0 vient d'entrer dans le bar
#2 vient d'entrer dans le bar
La personne #1 veut entrer
#1 vient d'entrer dans le bar
#0 a quitté le building !
La personne #3 veut entrer
#3 vient d'entrer dans le bar
La personne #4 veut entrer
La personne #5 veut entrer
La personne #6 veut entrer
La personne #7 veut entrer
La personne #8 veut entrer
La personne #9 veut entrer
#1 a quitté le building !
#4 vient d'entrer dans le bar
#2 a quitté le building !
#5 vient d'entrer dans le bar
#3 a quitté le building !
#6 vient d'entrer dans le bar
#4 a quitté le building !
#7 vient d'entrer dans le bar
#5 a quitté le building !
#8 vient d'entrer dans le bar
#6 a quitté le building !
#9 vient d'entrer dans le bar
#7 a quitté le building !
#8 a quitté le building !
#9 a quitté le building !
  
```

Le Join()

C'est le dernier mécanisme de synchronisation dont je parlerai. Il s'agit très simplement d'attendre la fin d'un autre thread afin de continuer le thread dans lequel le *Join()* est défini. Cela en fait une méthode bloquante qui pourrait vous causer des problèmes en Windows Forms.

Petit exemple :

Code : C#

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading;

namespace TestThread
{
    class Program
    {
        static void Main(string[] args)
        {
            Thread th = new Thread(new
            ParameterizedThreadStart(Afficher));

            Thread th2 = new Thread(new
            ParameterizedThreadStart(Afficher));

            th.Start("A");

            //On attend la fin du thread A avant de commencer le
            thread B.
            th.Join();
        }
    }
}
  
```

```

        th2.Start("B");

        Console.ReadKey();
    }

    static void Afficher(object texte)
    {
        for (int i = 0; i < 10000; i++)
        {
            Console.Write((string) texte);
        }
    }
}

```

Le Abort()

Bon, après avoir vu comment bien synchroniser ses threads, voyons ce que vous ne devez **PAS** faire !!! 🧐

Code : C#

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading;

namespace ThreadStop
{
    class Program
    {
        static void Main(string[] args)
        {
            Thread thread = new Thread(Test);

            thread.Start();
            Thread.Sleep(100);

            //On tue le processus. À NE PAS FAIRE !
            thread.Abort();

            Console.ReadKey();
        }

        public static void Test()
        {
            for (int i = 0; i < 10000; i++)
                Console.WriteLine(i);
        }
    }
}

```



Aux premiers abords, cela semble assez facile à faire, et semble sans grandes conséquences. En fait, vous avez probablement raison dans cet exemple. Cependant, ne prenez pas l'habitude de faire terminer vos threads si abruptement, car il se pourrait que cela vous cause des erreurs éventuellement. En effet, vous ne fermez pas votre ordinateur en débranchant la prise du mur, n'est-ce pas ? (🧐 N'est-ce pas ?). C'est le même principe ici. Si vous étiez en train de faire quelque chose de vraiment important, et que le thread principal choisirait ce moment pour arrêter le thread, l'application en entier pourrait devenir instable et planter. Puisqu'on ne veut pas cela, il vaut mieux utiliser le `Join()` et une variable de contrôle, comme dans l'exemple suivant :

Code : C#

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading;

namespace ThreadStop
{
    class Program
    {
        private static bool _continuer = true;

        static void Main(string[] args)
        {
            Thread thread = new Thread(Test);

            thread.Start();
            Thread.Sleep(100);

            //On demande au thread de s'arrêter au prochain passage
            //d'un moment qui semble naturel.
            _continuer = false;

            //On attend que le thread se termine.
            thread.Join();

            Console.ReadKey();
        }

        public static void Test()
        {
            //On fait 10 000 itérations, tant et aussi longtemps que
            //l'on peut continuer (variable de contrôle).
            for(int i = 0; i < 10000 && _continuer; i++)
                Console.WriteLine(i);
        }
    }
}

```

Et voilà, on se sent toujours mieux quand on fait quelque chose de bien, non ? Comme ça, si le thread a besoin de temps pour bien terminer ses opérations (appeler quelques *Dispose()*, ou fermer des connexions TCP), il le pourra. Utilisez donc les *Join()* et pas les *Abort()*. Les *Abort()*, c'est mal 😞...



Lorsque vous fermez l'application, Windows tend à simplement appeler *Abort()* sur les threads en fonction. Ce comportement est hérité du Win32 dont le .NET recouvre à l'aide de *wrappers* (il s'agit d'une longue histoire, croyez-moi). C'est aussi pourquoi on évitera autant que possible l'usage de threads indépendants, car ils sont moins contrôlables. Prévoyez donc attendre les threads actifs lorsque vous quitter votre application, car comme nous l'avons dit, les *Abort()*, c'est mal. Quoique avec le Garbage Collector de .NET.... **NON, NON, C'EST MAL !**

Nous sommes maintenant prêts à aborder le sujet du multi-tâche en Windows Forms ! Je vous montrerai comment éviter que cela ne vire en catastrophe, ne craignez rien.

Les threads avec les Windows Forms

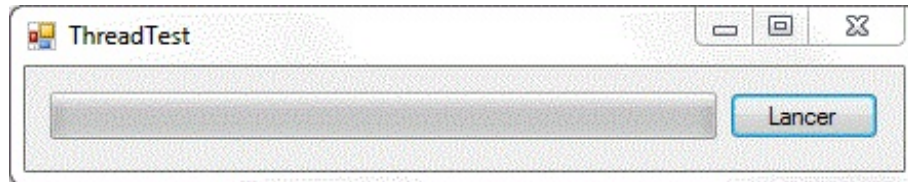
Lorsque viendra le temps de faire du réseau, il sera important de respecter la sécurité interne des ressources. Il sera très souvent intéressant de modifier un objet **Windows Forms** avec les données issues d'un autre thread. C'est notamment le cas en réseau, si on fait un programme de *ch@t*, par exemple.

Rappelez-vous, un peu plus haut, quand je vous ai dit qu'un thread, même s'il est séparé, a accès aux membres de sa classe ? Je

ne vous ai pas menti. Mais dans le cas des objets Windows Forms, ceux-ci ne sont pas bâties **thread-safe** et donc que .NET limite leurs possibilités en multi-thread. Cela signifie que, malheureusement, vous ne pourrez accéder à leurs propriétés qu'en lecture seule si vous ne faites pas parti du même thread. Je vous donne un exemple :

J'ai construit une Windows Forms très simple n'ayant que deux éléments :

- Une ProgressBar
- Un Bouton



J'ai ajouté le code suivant dans le fichier *Form1.cs*.

Code : C#

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;
using System.Threading;

namespace ThreadWForms
{
    public partial class Form1 : Form
    {
        private Random rand = new Random((int)DateTime.Now.Ticks);

        private int[] tableau = new int[100000000];

        public Form1()
        {
            InitializeComponent();

            //On génère un tableau d'entiers aléatoires.
            for (int i = 0; i < tableau.Length; i++)
            {
                tableau[i] = rand.Next(50000);    //...Boucle très
simple, avec méthode Random très simple.
            }

            public void Selection()
            {
                //On va simplement compter les nombres du tableau
inférieurs à 500.
                int total = 0;

                for (int i = 0; i < tableau.Length; i++)
                {
                    if (tableau[i] < 500)
                    {
                        total++;
                    }

                    //Puis, on incrémente le ProgressBar.
                    pgbThread.Value = (int)(i / (double)tableau.Length *
100);
                }
            }
        }
    }
}
```

```

    }
}

private void btnLancer_Click(object sender, EventArgs e)
{
    //On crée le thread.
    Thread t1 = new Thread(new ThreadStart(Selection));

    //Puis on le lance !
    t1.Start();
}
}

```



Ce code n'est pas bon, bien que logique. Cela produira cette erreur :

```

for (int i = 0; i < tableau.Length; i++)
{
    if (tableau[i] < 500)
    {
        total++;
    }

    //Puis, on incrémente le ProgressBar.
    pgbThread.Value = (int)(i / (double)tableau.Length * 100);
}

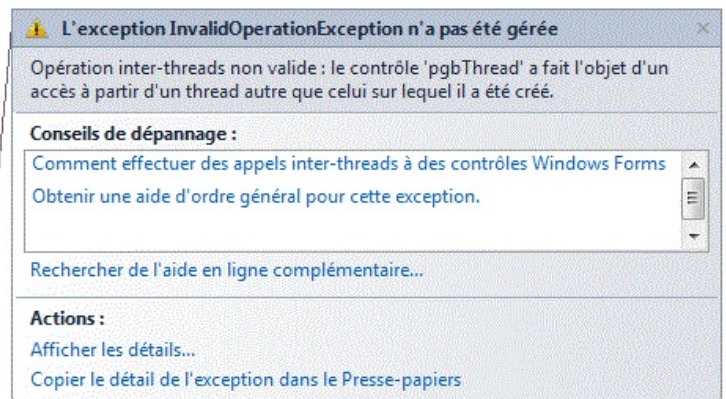
```

```

void btnLancer_Click(object sender, EventArgs e)
{
    //On crée le thread.
    Thread t1 = new Thread(new ThreadStart(Selection));

    //Puis on le lance !
    t1.Start();
}

```



Alors, je vous l'avais pas dit que ça ne marcherait pas ? Il existe heureusement une façon bien simple de contrer ce problème, et c'est de passer par les *delegates* ! En effet, car si ceux-ci peuvent être passés en paramètres, il peuvent aussi servir à exécuter des opérations sur un thread différent !

Code : C#

```

using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;
using System.Threading;

namespace ThreadWForms
{
    public partial class Form1 : Form
    {
        private Random rand = new Random((int)DateTime.Now.Ticks);

        private int[] tableau = new int[500000];

        //On crée notre delegate.
        public delegate void MontrerProgres(int valeur);

        bool termine = true;
    }
}

```

```

    public Form1()
    {
        InitializeComponent();

        //On génère un tableau d'entiers aléatoires.
        for (int i = 0; i < tableau.Length; i++)
        {
            tableau[i] = rand.Next(50000);    //...Boucle très
simple, avec méthode Random très simple.
        }
    }

    public void Selection()
    {
        //On va simplement compter les nombres du tableau
inférieurs à 500.
        int total = 0;

        for (int i = 0; i < tableau.Length; i++)
        {
            if (tableau[i] < 500)
            {
                total++;
            }

            //Puis, on incrémente le ProgressBar.
            int valeur = (int)(i / (double)tableau.Length *
100);

            //On achète la paix, on entoure notre Invoke d'un
try...catch !
            try
            {
                //On invoque le delegate pour qu'il effectue la
tâche sur le temps
                //de l'autre thread.
                Invoke((MontrerProgres)Progres, valeur);
            }
            catch (Exception ex) { return; }
        }

        termine = true;
    }

    private void btnLancer_Click(object sender, EventArgs e)
    {
        //Petite sécurité pour éviter plusieurs threads en même
temps.
        if (termine)
        {
            //On crée le thread.
            Thread t1 = new Thread(new ThreadStart(Selection));

            termine = false;

            //Puis on le lance !
            t1.Start();
        }
    }

    public void Progres(int valeur)
    {
        //On met la valeur dans le contrôle Windows Forms.
        pgbThread.Value = valeur;
    }
}

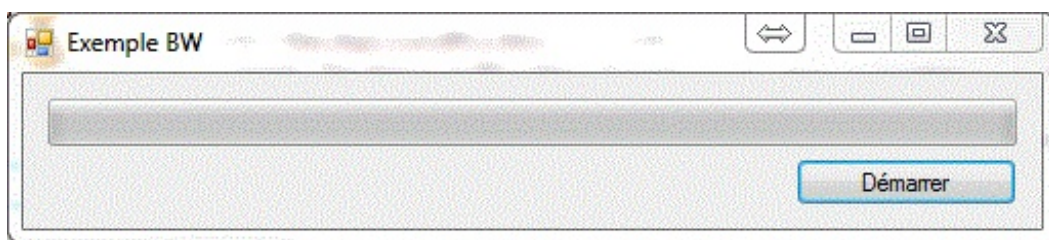
```

Petites explications : Un *Invoke* sert à demander à l'autre thread de s'occuper d'une action dans un moment libre. C'est l'équivalent d'envoyer un email à un webmestre pour qu'il corrige une erreur sur sa page web. Le webmestre, dès qu'il le pourra, s'occupera de corriger l'erreur. Il s'agit du même principe ! Cela permet de contourner le manque *thread-safe* des contrôles Windows Forms, car c'est le thread propriétaire qui finit par effectuer l'action.

Aussi, vous avez peut-être fait la grimace en apercevant la ligne 55. Il y a un cast d'un delegate juste avant le nom de la méthode. Cela évite d'avoir à créer un delegate. Je pourrais très bien remplacer cette ligne par ceci : `Invoke (new MontrerProgres (Progres), valeur);`.

BackgroundWorker

La classe `BackgroundWorker` fournit un environnement d'exécution multitâche très sécuritaire, mais un peu limité à mon goût. Cependant, je vais quand même vous montrer comment l'utiliser. L'objet `BackgroundWorker` se trouve dans la barre d'outils dans la catégorie Composants. Il s'agit d'un petit objet pas très personnalisable qui possède très peu de paramètres et d'événements. Je vous montre par un exemple comment l'utiliser. J'ai fait un simple projet Windows Forms dans Visual Studio qui comporte une `ProgressBar` et un bouton de départ, tout comme l'exemple précédent.



Code : C#

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;
using System.Threading;

namespace Background_WorkerEx
{
    public partial class Form1 : Form
    {
        private bool _etat = false;

        public Form1 ()
        {
            InitializeComponent();

            //On demande à ce que le BackgroundWorker supporte le
            rapport de progrès et l'annulation.
            bwProgress.WorkerReportsProgress = true;
            bwProgress.WorkerSupportsCancellation = true;

            //On abonne le BackgroundWorker aux événements requis.
            bwProgress.DoWork+=new
            DoWorkEventHandler (bwProgress_DoWork);
            bwProgress.ProgressChanged+=new
            ProgressChangedEventHandler (bwProgress_ProgressChanged);
            bwProgress.RunWorkerCompleted+=new
            RunWorkerCompletedEventHandler (bwProgress_RunWorkerCompleted);
        }

        private void bwProgress_DoWork(object sender,
            DoWorkEventArgs e)
        {
```

```

        int i = 0;

        //Tant et aussi longtemps que la barre n'a pas atteint
le 100% et qu'on
        //ne demande pas à annuler...
        while (i < 100 && !bwProgress.CancellationPending)
        {
            //On attend 150ms.
            Thread.Sleep(150);

            //On retrouve la valeur la plus petite entre 100 et
i + 3.
            i = Math.Min(100, i + 3);

            //On rapporte le progrès fait.
            bwProgress.ReportProgress(i);
        }
    }

    private void btnStart_Click(object sender, EventArgs e)
    {
        //Le bouton joue le rôle de démarrage comme
d'annulation selon la situation.
        if (!_etat)
        {
            bwProgress.RunWorkerAsync();
            btnStart.Text = "Annuler";
        }
        else
        {
            bwProgress.CancelAsync();
            btnStart.Text = "Démarrer";
        }

        _etat = !_etat;
    }

    private void bwProgress_ProgressChanged(object sender,
ProgressChangedEventArgs e)
    {
        //On fait avancer la ProgressBar.
        pgProgress.Value = e.ProgressPercentage;
    }

    private void bwProgress_RunWorkerCompleted(object sender,
RunWorkerCompletedEventArgs e)
    {
        //Lorsque c'est terminé, on affiche un message
indiquant la fin de l'activité.
        MessageBox.Show("Le BackgroundWorker a terminé");
    }
}

```



Tout ce qui a été défini dans le constructeur du formulaire aurait pu être mis dans le Designer graphique. Je l'ai mis là pour vous montrer quelles propriétés et événements ont été affectés.

Donc avec le BackgroundWorker, le multi-tâche en Windows Forms devient très facile comme vous pouvez le voir. Tous les événements appelés seront exécutés dans le thread principal, éliminant ainsi l'utilisation de la commande `Invoke`. Magique n'est-ce pas ? 🧙

Cela conclut ce tutoriel. Il y a plusieurs parties des threads et des delegates que je n'ai pas couvert parce que je ne les considère pas nécessaires. Allez faire un tour du côté de MSDN pour plus d'explications et d'exemples ! Je vous met sur quelques pistes, comme la programmation réseau, les méthodes anonymes et les impressions en threads séparés.

Il n'y a jamais de fin à ce que le .NET peut faire ! Les threads n'étaient que la pointe du iceberg. Visitez MSDN si vous désirez plus d'information et si vous vous posez une question sans réponse, je vous rappelle qu'il y a un Forum sur le Site du Zéro pour demander.

Partager



Ce tutoriel a été corrigé par les [zCorrecteurs](#).